Smalltalk Implementation:
Optimization Techniques

Prof. Harry Porter
Portland State University

Optimization Ideas

- Just-In-Time (JIT) compiling
  When a method is first invoked, compile it into native code.
- Caching the Method Dictionary
  Method Look-up will be speeded up.
- Inline Method Sending
  Will turn many SENDs into native CALL instructions
- Use the hardware calling stack
  MethodContexts → activation records allocated on a stack
- Code the VM directly in Smalltalk
  Automatic translation into "C"

Misc Points

Porting the Smalltalk Interpreter

The virtual machine is implemented in Smalltalk!

Using a subset of Smalltalk, called "*Slang*"

The image also includes a translator / compiler

Slang→ "C"

Steps to porting:

- **Produce automatically generated interpreter in "C"**
- **Hand-code the machine-dependent parts in "C"**
- **Compile**
- **Use any existing image**

Misc Points

Porting Images

Each VM executes the same bytecodes.
***Any* image can be executed on by *any* VM.**

EXAMPLE: An image produced on *MAC OS X* can be executed on *Windows*.

Porting Code Fragments

Misc Points

Porting Images

Each VM executes the same bytecodes.
Any image can be executed on by *any* VM.

EXAMPLE: An image produced on *MAC OS X*
can be executed on *Windows*.

Porting Code Fragments

Also, code fragments can be *filed out*
… and *filed in* to another image

Will it work?

The Smalltalk language is uniform.
What pre-existing classes does the code use?

Misc Points

Hash Values

Some classes rely on "hash values".

Dictionary, *Set*, etc.

Every object must be capable of providing its hash value:

```
i := x hashValue.
```

Misc Points

Hash Values

Some classes rely on "hash values".

Dictionary, *Set*, etc.

Every object must be capable of providing its hash value:

```
i := x hashValue.
```

Two objects can contain exactly the same values.
They differ only in where they are in memory
…and GC will move objects around

Misc Points

Hash Values

Some classes rely on "hash values".
Dictionary, *Set*, etc.

Every object must be capable of providing its hash value:

```
i := x hashValue.
```

Two objects can contain exactly the same values.
They differ only in where they are in memory
…and GC will move objects around

Need special VM support for hash values!

- Each object contains a hash value.
- 12 bits
- Stored in it header
- Initialized when the object is created

Optimizations to the Interpreter

Virtual Machine

Does not match underlying hardware well

Examples:

- OOP/SmallInteger Tagging
- Registers versus Stacks in Context objects

Bytecodes vs. Machine Instructions

The bytecodes are interpreted

- Fetch-decode-execute done at two levels.

Difficult to optimize bytecodes

- Bytecodes are complex operations
  - Corresponding to several machine level instructions

"Just in Time" Compiling

Translate bytecodes into native machine language

… and execute them directly

Do it "on the fly"

… on individual methods

Source → bytecodes → machine instructions

When the method is first invoked…

- Call the JIT compiler
- Translate bytecodes to native instructions
- Save the native code for next time.

"Just in Time" Compiling

Benefits:

- Optional
  Compatible with existing system
- Still have bytecodes
  (for the debugging tools)
- Can perform many optimizations at the native code level
- Can do it just to frequently invoked methods
- Running out of memory?
  Throw away some of the compiled methods

"Just in Time" Compiling

Problem:

Activation records are user-visible

MethodContexts, ***BlockContexts***

Activation record contains a pointer to the current bytecode

"instructionPointer" = "Program Counter (PC)"

Used by the debugging tools!

Solution:

"Just in Time" Compiling

Problem:

Activation records are user-visible

MethodContexts, ***BlockContexts***

Activation record contains a pointer to the current bytecode

"instructionPointer" = "Program Counter (PC)"

Used by the debugging tools!

Solution:

Whenever an activation record becomes user-visible…

Map the native code PC back into a bytecode PC

Allocating Contexts on the Hardware Stack

The hardware supports stacks & procedure CALLs well.

"*stack frame*" = "*activation record*"

Allocating Contexts on the Hardware Stack

The hardware supports stacks & procedure CALLs well.

"***stack frame***" = "***activation record***"

Smalltalk VM…

linked list of ***Context*** objects

Want to use the hardware stack

Want to store each ***Context*** as a "stack frame"

Contexts are usually allocated in LIFO (stack) order.

Not usually accessed as an object

Allocating Contexts on the Hardware Stack

The hardware supports stacks & procedure CALLs well.

"***stack frame***" = "***activation record***"

Smalltalk VM…

linked list of ***Context*** objects

Want to use the hardware stack

Want to store each ***Context*** as a "stack frame"

Contexts are usually allocated in LIFO (stack) order.

Not usually accessed as an object

Exception: When debugging, the debugger

Asks for a pointer to the current context

Treats it as (non-stack) data

The Idea:

Allocating Contexts on the Hardware Stack

The hardware supports stacks & procedure CALLs well.

"***stack frame***" = "***activation record***"

Smalltalk VM…

linked list of ***Context*** objects

Want to use the hardware stack

Want to store each ***Context*** as a "stack frame"

Contexts are usually allocated in LIFO (stack) order.

Not usually accessed as an object

Exception: When debugging, the debugger

Asks for a pointer to the current context

Treats it as (non-stack) data

The Idea:

Store stack frames on hardware stack, not as objects.

When a pointer is generated to the current context…

Convert the stack frame into a real object.

Details

Converting a stack frame into a real object…

Allocate a new ***Context*** object and fill in its fields

Convert the program counter (PC)

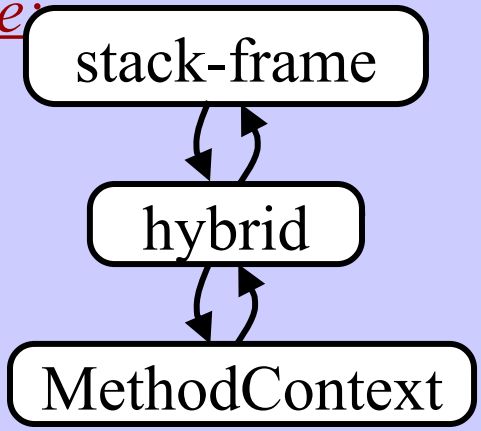
absolute address → byte offset into a ***CompiledMethod*** object

Contexts point to other ***Context***s

But other ***Context***s are still on hardware stack

Convert all frames into Objects…? No!

The Technique:



Caching the Method Dictionary

Method Lookup:

Given:
- the receiver's class
- the message selector

Find:
- the right ***CompiledMethod***

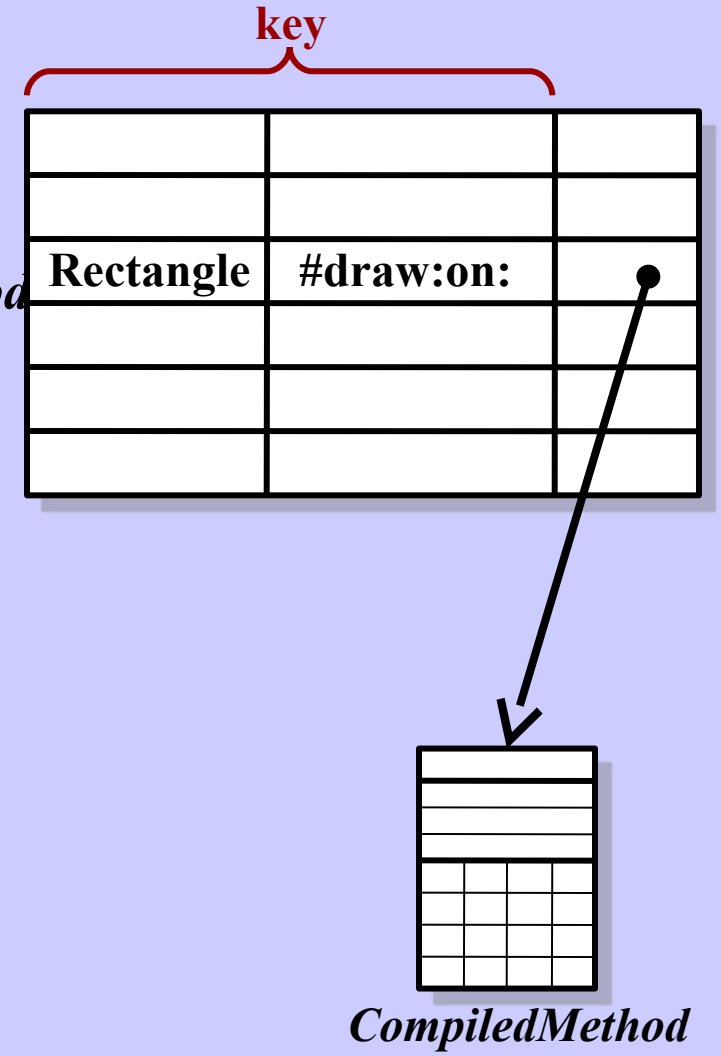
The Idea:

Caching the Method Dictionary

Method Lookup:

Given:
- the receiver's class
- the message selector

Find:
- the right *CompiledMethod*



The Idea:

Use a Hash Table

Maintained by the VM

(it is not an object)

Not in the hash table?
- Do a full method lookup
- Add an entry to the hash table

Inline Method Caching

Assume methods are compiled into native code.

Code to send a message:

```
< code to push receiver >
< code to push args >
CALL MessageSend (arg = selector)
```

A machine-language CALL instruction

A routine that searches for the proper method/routine and then calls it.

Inline Method Caching

Assume methods are compiled into native code.

Code to send a message:

```
< code to push receiver >
< code to push args >
CALL MessageSend (arg = selector)
```

The Idea:

- Upon locating the correct routine…
  Replace the CALL to the "MessageSend" routine
  … with a CALL straight to the native code routine!
- Next time we execute the above code,
  we CALL the right routine immediately.
- Gradually all message sends are replaced with
  native code CALL instructions.

Inline Method Caching

Problem:

Dynamic Look-Up

The receiver's class determines which method to invoke.
Different class? → Different method!

Assumption:

Approach:

Inline Method Caching

Problem:

Dynamic Look-Up

The receiver's class determines which method to invoke.
Different class? → Different method!

Assumption:

Any particular SEND will invoke the same method
... almost always!

Approach:

Inline Method Caching

Problem:

Dynamic Look-Up

The receiver's class determines which method to invoke.

Different class? → Different method!

Assumption:

Any particular SEND will invoke the same method

... almost always!

Approach:

At the beginning of each method:

- Check the class of the receiver
- If it is what this method expects

… continue with this method.

- If the receiver has the wrong class…
- Perform a full method lookup.
- Overwrite the CALL (to jump to the correct method next time)
- Jump to the correct method.

Effectiveness of Optimizations

	space	time
Straight interpreter	1.0	1.0
Compiler	2.3	.69
Compiler w/ inline caching	3.4	.62
Compiler w/ peephole optimizer	5.0	.56
Compiler w/ inline caching w/ optimizer	5.0	.51